A review on fixed threshold based and adaptive threshold based auto-scaling techniques in cloud computing

Asif Ali Khan 1, CH Vidhyadhari 2, Suresh Kumar 3

1 KGRCET, CSE (AI&ML) Department, Hyderabad, India
2 Department of IT, GRIET, Hyderabad, Telangana, India
3 Lovely Professional University, Phagwara, Punjab, India

Abstract. Cloud computing has evolved as an effective technology in the past few years. It is playing an important role in information technology. The main characteristics and challenges of cloud computing are 100% availability and Scalability. To achieve these two things we have concepts like Load balancing and auto-scaling in the cloud. In this paper, we will discuss auto-scaling and the two different ways to achieve auto-scaling. The commonly used auto-scaling technique is fixed threshold-based auto-scaling and it is further modified as adaptive threshold-based auto-scaling. Auto-scaling can scale up or down the cloud resources based on demand, which means the virtual machines are automatically launched or deleted based on the requirement. As Auto-scaling provides better fault tolerance, better availability, and cost management, it improves an organization’s service level agreements.

1 Introduction to Auto-Scaling

1.1 Key Components

1.1.1 Metrics and Monitoring:
1.1 Scaling Policies:

Scaling policies define the conditions under which the system should scale in or out. These policies are rules based on the monitored metrics. For example, a scaling policy might state that if CPU utilization exceeds a certain threshold for a specified duration, then add more resources (scale out). Conversely, if the utilization drops below another threshold, remove resources (scale in).

1.1.3 Scaling Actions

Scaling actions are the actual adjustments made to the system in response to scaling policies. For scaling out, actions might include launching additional instances, adding more containers, or increasing the capacity of existing resources. For scaling in, actions might involve terminating instances, reducing container count, or decreasing the capacity of existing resources.

1.1.4 Infrastructure Orchestration

Auto-scaling often involves coordination with infrastructure orchestration tools. In cloud environments, this may include interacting with services like AWS Auto Scaling Groups, Azure Virtual Machine Scale Sets, or Google Cloud Instance Groups. These tools automate the deployment and termination of instances or resources based on scaling actions.

1.2 Types of Auto-Scaling

1.2.1 Horizontal Scaling (Scale-Out/In)

Increases or decreases the number of instances or resources to handle varying workloads. Common in cloud environments where you add or remove virtual machines, containers, or serverless functions.
1.1.2 Vertical Scaling (Scale Up/Down):

Adjust the capacity of individual instances by adding or removing resources such as CPU, memory, or storage. Could be limited by the maximum resources a single instance can have [2].

2 Comparing fixed threshold-based auto-scaling and adaptive threshold-based auto-scaling techniques

2.1 Dynamic Adaptability

2.1.1 Adaptive Thresholds:

Adjusts thresholds dynamically based on the observed behavior of the system. This allows the auto-scaler to adapt to changing workload patterns, making it more responsive to fluctuations in demand.

2.1.2 Fixed Thresholds:

Static thresholds might lead to overprovisioning or underprovisioning in scenarios where the workload patterns change over time. Fixed thresholds may not respond well to unexpected spikes or drops in demand.
2.2 Optimized Resource Utilization

2.2.1 Adaptive Thresholds:

By adapting to the actual workload patterns, adaptive threshold-based auto-scaling can potentially optimize resource allocation, ensuring that the system scales in and out at the most appropriate times.

2.2.2 Fixed Thresholds:

Fixed thresholds may not always align with the actual resource requirements of the application, leading to suboptimal resource utilization.

2.3 Reduced Manual Tuning

2.3.1 Adaptive Thresholds:

Requires less manual tuning compared to fixed thresholds. The auto-scaler can learn and adjust thresholds based on historical data, reducing the need for administrators to set and modify threshold values.

2.3.2 Fixed Thresholds:

Require manual tuning to set thresholds initially and may need frequent adjustments to accommodate changes in workload patterns.

2.4 Handling Seasonal or Periodic Workloads:

2.4.1 Adaptive Thresholds:

Well-suited for applications with seasonal or periodic variations in workload. The auto-scaler can adapt to these variations and adjust thresholds accordingly.

2.4.2 Fixed Thresholds:

May struggle to adapt to changing workload patterns and might not effectively handle seasonal variations without frequent manual adjustments.

2.5 Learning and Adaptation

2.5.1 Adaptive Thresholds:

Incorporates a learning component that adapts to the system's behavior over time. This learning capability enables the auto-scaler to make more informed decisions.
2.5.2 Fixed Thresholds:

Lack the learning component, making them less adaptive to changing conditions.

2.6 Cost Efficiency:

2.6.1 Adaptive Thresholds:

Can contribute to cost efficiency by scaling resources more accurately based on actual demand, potentially reducing unnecessary overprovisioning.

2.6.2 Fixed Thresholds:

May lead to overprovisioning during periods of low demand or under-provisioning during peaks, impacting cost efficiency.

It's important to note that the choice between adaptive and fixed threshold-based auto-scaling depends on the specific characteristics of your application and workload. In some cases, a combination of both approaches and periodic adjustments to fixed thresholds might be appropriate. Regular monitoring, evaluation, and fine-tuning based on the changing nature of your system are essential for optimal auto-scaling performance.

3 Achieving scalability using fixed threshold-based auto-scaling

Fixed threshold-based auto-scaling is a straightforward approach where scaling decisions are made based on predefined static thresholds. This method involves setting specific values for metrics such as CPU utilization, memory usage, or other relevant indicators. When the monitored metric crosses a certain threshold, the auto-scaling system triggers scaling actions to add or remove resources.

Fixed threshold-based auto-scaling is suitable for scenarios where workload patterns are relatively stable and predictable. Regular monitoring and adjustment of thresholds are essential to ensure that the auto-scaling system aligns with the changing needs of the application.

Here is a step-by-step procedure for implementing a fixed threshold-based auto-scaling algorithm:

3.1 Identify Metrics for Monitoring

Determine which metrics are critical for your application's performance. Common metrics include CPU utilization, memory usage, network traffic, or any other relevant performance indicators.
3.2 Set Thresholds

Establish fixed thresholds for each monitored metric. These thresholds will determine when the auto-scaling algorithm should trigger scaling actions. For example, set a threshold of 80% for CPU utilization.

3.3 Define Scaling Policies

Establish scaling policies based on the defined thresholds. Determine what actions should be taken when a metric exceeds or falls below its threshold.

3.3.1 Scale Out Policy:

Increase resources (e.g., add instances) when a metric exceeds its upper threshold.

3.3.2 Scale in Policy:

Decrease resources (e.g., remove instances) when a metric falls below its lower threshold.

3.4 Choose Cool down Periods

Implement cool-down periods to prevent rapid and consecutive scaling actions. A cool-down period is a time interval during which the auto-scaling system ignores additional triggers to ensure system stability.

3.5 Monitor Metrics

Continuously monitor the selected metrics in real time. This can be done using monitoring tools or APIs provided by your cloud provider.

3.6 Implement Scaling Actions

When a metric crosses a threshold, execute the predefined scaling action.

I. If CPU utilization exceeds 80%, add more instances to the fleet.
II. If CPU utilization drops below 60%, remove instances from the fleet.

3.7 Test the Auto-Scaling System

Conduct thorough testing to ensure that the auto-scaling system responds appropriately to changing workloads and adheres to the defined thresholds and policies.

3.8 Set Up Alerting

Implement alerting mechanisms to notify administrators or relevant stakeholders when thresholds are crossed. This ensures timely awareness of scaling actions.
3.9 Evaluate and Fine-Tune

Regularly evaluate the performance of the auto-scaling system in a production environment. Analyse the effectiveness of the chosen thresholds and policies. If necessary, fine-tune the thresholds and policies based on observed behavior and changing workload patterns.

3.10 Documentation

Document the implemented fixed threshold-based auto-scaling strategy. Include details on thresholds, policies, cool down periods, and any other relevant information. This documentation aids in understanding and maintaining the auto-scaling system.

4. Fixed threshold-based auto-scaling algorithm in Python programming language

Implementing a threshold-based auto-scaling algorithm depends on the specific environment and tools you are using. Below is a simplified example in Python that demonstrates a basic threshold-based auto-scaling algorithm. This example assumes a simple scenario where we monitor CPU utilization and trigger scaling actions based on predefined thresholds.

```python
class AutoScaler:
    def __init__(self, upper_threshold, lower_threshold):
        self.upper_threshold = upper_threshold
        self.lower_threshold = lower_threshold
        self.is_scaling_out = False
        self.is_scaling_in = False

    def monitor_cpu_utilization(self, current_cpu_utilization):
        if current_cpu_utilization > self.upper_threshold:
            self.scale_out()
        elif current_cpu_utilization < self.lower_threshold:
            self.scale_in()

    def scale_out(self):
        if not self.is_scaling_out:
            print("Scaling out: Adding more instances...")
            # Implement logic to add more instances or scale resources
            self.is_scaling_out = True
            self.is_scaling_in = False

    def scale_in(self):
        if not self.is_scaling_in:
            print("Scaling in: Removing instances...")
            # Implement logic to remove instances or scale down resources
            self.is_scaling_in = True
            self.is_scaling_out = False
```

def reset_scaling_flags(self):
    self.is_scaling_out = False
    self.is_scaling_in = False

# Example Usage:
if __name__ == '__main__':
    # Set your upper and lower CPU utilization thresholds
    upper_threshold = 80  # Example: Scale out if CPU utilization exceeds 80%
    lower_threshold = 60  # Example: Scale in if CPU utilization drops below 60%
    auto_scaler = AutoScaler(upper_threshold, lower_threshold)
    # Simulate monitoring CPU utilization over time
    cpu_utilization_data = [75, 85, 90, 55, 70, 95, 50, 75, 80]
    for cpu_utilization in cpu_utilization_data:
        auto_scaler.monitor_cpu_utilization(cpu_utilization)
    # Reset scaling flags after monitoring is complete
    auto_scaler.reset_scaling_flags()

5. Adaptive threshold-based auto-scaling

An adaptive threshold-based auto-scaling technique adjusts the scaling thresholds dynamically based on historical data or the current state of the system. This allows the auto-scaler to adapt to changing workload patterns and optimize resource allocation [15].

Below is a basic algorithm for an adaptive threshold-based auto-scaling technique:

```python
class AdaptiveAutoScaler:
    def __init__(self, initial_threshold, learning_rate):
        self.current_threshold = initial_threshold
        self.learning_rate = learning_rate
        self.is_scaling_out = False
        self.is_scaling_in = False

    def monitor_metric(self, current_metric_value):
        if current_metric_value > self.current_threshold:
            self.scale_out()
        elif current_metric_value < self.current_threshold:
            self.scale_in()

    def scale_out(self):
        if not self.is_scaling_out:
            print('Scaling out: Adding more instances...')
            # Implement logic to add more instances or scale resources
            self.is_scaling_out = True
            self.is_scaling_in = False

    def scale_in(self):
        # Implement logic to remove instances or scale back resources
        self.is_scaling_out = False
        self.is_scaling_in = True
```

5. Adaptive threshold-based auto-scaling
if not self.is_scaling_in:
    print("Scaling in: Removing instances...")
    # Implement logic to remove instances or scale down resources
    self.is_scaling_in = True
    self.is_scaling_out = False

def adapt_threshold(self, observed_metric_values):
    if len(observed_metric_values) > 0:
        average_metric_value = sum(observed_metric_values) / len(observed_metric_values)
        self.current_threshold = (1 - self.learning_rate) * self.current_threshold + self.learning_rate * average_metric_value

def reset_scaling_flags(self):
    self.is_scaling_out = False
    self.is_scaling_in = False

# Example Usage:
if __name__ == "__main__":
    initial_threshold = 80  # Set an initial threshold
    learning_rate = 0.1  # Set a learning rate
    adaptive_scaler = AdaptiveAutoScaler(initial_threshold, learning_rate)
    # Simulate monitoring metric over time
    observed_metric_data = [75, 85, 90, 55, 70, 95, 50, 75, 80]
    for metric_value in observed_metric_data:
        adaptive_scaler.monitor_metric(metric_value)  # Adapt the threshold based on observed metric values
        adaptive_scaler.adapt_threshold(observed_metric_data)  # Reset scaling flags after monitoring is complete

6. Conclusion

Adaptive threshold-based auto-scaling and fixed threshold-based auto-scaling each have their advantages and disadvantages. The choice between them depends on the characteristics of your workload, the dynamic nature of your application, and your specific scaling requirements. It's important to note that the choice between adaptive and fixed threshold-based auto-scaling depends on the specific characteristics of your application and workload. In some cases, a combination of both approaches and periodic adjustments to fixed thresholds might be appropriate. Regular monitoring, evaluation, and fine-tuning based on the changing nature of your system are essential for optimal auto-scaling performance.
References


